CSCI-344
Programming Language Concepts (Section 3)

Lecture 28

Reduction Semantics for Control Structures

Instructor: Hossein Hojjat

November 18, 2016

Where we are

Done:

- Control Operators
  - `(break)`
  - `(continue)`
  - `(return` *exp*`)`
  - `(try-catch` *exp* *exp*`)`
  - `(throw` *exp*`)`

This session:

- Reduction Semantics for Control Structures

Simple Expression Language

- Consider a very simple language for addition

$$\begin{array}{lcll} \mathsf{Exp} & = & \mathrm{LIT} & (\mathsf{Value}) \\ & \mid & \mathrm{VAR} & (\mathsf{Name}) \\ & \mid & \mathrm{ADD} & (\mathsf{Exp}, \mathsf{Exp}) \end{array}$$

Operational Semantics

- Judgment $\langle e, \rho \rangle \Downarrow n$ means e evaluates to n in environment ρ
- Judgment specifies the entire transition from a configuration to a final value
- $\mathrm{ADD}(e_1, e_2)$: is e_1 evaluated first or e_2?

$$\frac{}{\langle \mathrm{LIT}(n), \rho \rangle \Downarrow n} \qquad \frac{}{\langle \mathrm{VAR}(x), \rho \rangle \Downarrow \rho(x)}$$

$$\frac{\langle e_1, \rho \rangle \Downarrow n_1 \quad \langle e_2, \rho \rangle \Downarrow n_2}{\langle \mathrm{ADD}(e_1, e_2), \rho \rangle \Downarrow n_1 + n_2}$$

Big-step vs. small-step Operational Semantics

- Big-step operational semantics: transition encodes all computation steps
$$\langle e, \rho \rangle \Downarrow n$$
- Small-step operational semantics: transition encodes only one step of computation
$$\langle e, \rho \rangle \rightarrow \langle e', \rho \rangle$$

Big-step vs. small-step Operational Semantics

Evaluation of an expression e in environment ρ:

- Big-step: $\langle e, \rho \rangle \Downarrow n$
- Small-step: $\langle e, \rho \rangle \to \langle e', \rho \rangle \to \langle e'', \rho \rangle \to \cdots \to n$

Big-step Semantics

- Corresponds to the transitive closure of small steps
- Evaluation skips over intermediate steps:
  programs without final configurations (infinite loops, errors)
  look the same

Small-step Operational Semantics

$$\frac{}{\langle \mathrm{LIT}(n), \rho\rangle \to n} \qquad \frac{}{\langle \mathrm{VAR}(x), \rho\rangle \to \rho(x)}$$

$$\frac{\langle e_1, \rho\rangle \to \langle e_1', \rho\rangle}{\langle \mathrm{ADD}(e_1, e_2), \rho\rangle \to \langle \mathrm{ADD}(e_1', e_2), \rho\rangle}$$

$$\frac{\langle e_2, \rho\rangle \to \langle e_2', \rho\rangle}{\langle \mathrm{ADD}(\mathrm{LIT}(n_1), e_2), \rho\rangle \to \langle \mathrm{ADD}(\mathrm{LIT}(n_1), e_2'), \rho\rangle}$$

$$\frac{}{\langle \mathrm{ADD}(\mathrm{LIT}(n_1), \mathrm{LIT}(n_2)), \rho\rangle \to n_1 + n_2}$$

- Fixed evaluation order
- Example: $\mathrm{ADD}(\ \underbrace{\mathrm{ADD}(\mathrm{LIT}(2), \mathrm{LIT}(5))}_{\text{first}}\ ,\ \underbrace{\mathrm{ADD}(\mathrm{LIT}(6), \mathrm{LIT}(3))}_{\text{second}}\ )$

Holes

- Operational Semantics: how to compute a program on some abstract machine
- How does the abstract machine compute a step in small-step semantics?
- To evaluate $\mathrm{ADD}(e_1, e_2)$ a small-step machine needs to:

1. Evaluate e_1,
2. Remember to return and evaluate e_2,
3. Finally compute the result of ADD.

Holes

- Small-step machine maintains a “TO-DO” stack of contexts (continuations)
- Context is an expression with a **hole** in the place of a sub-expression
- Example: to evaluate

  $$\mathrm{ADD}\left(\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(2)), \mathrm{LIT}(3)\right)$$

  machine needs to compute

  $$\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(2))$$

  in the context

  $$\mathrm{ADD}(\bullet, \mathrm{LIT}(3))$$

Example

Expression	Context Stack
$\mathrm{ADD}\,(\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(3)), \mathrm{LIT}(5))$	[ ]

Example

Expression	Context Stack
$\mathrm{ADD}\,(\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(3)), \mathrm{LIT}(5))$	$[\,]$
$\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(3))$	$\mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$

Example

Expression	Context Stack
$\mathrm{ADD}\,(\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(3)), \mathrm{LIT}(5))$	$[\,]$
$\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(3))$	$\mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
$\mathrm{LIT}(1)$	$\mathrm{ADD}(\bullet, \mathrm{LIT}(3)) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$

Example

Expression	Context Stack
$\mathrm{ADD}\,(\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(3)), \mathrm{LIT}(5))$	$[\,]$
$\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(3))$	$\mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
$\mathrm{LIT}(1)$	$\mathrm{ADD}(\bullet, \mathrm{LIT}(3)) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
1	$\mathrm{ADD}(\bullet, \mathrm{LIT}(3)) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$

Example

Expression	Context Stack
$\mathrm{ADD}\,(\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(3)), \mathrm{LIT}(5))$	$[\,]$
$\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(3))$	$\mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
$\mathrm{LIT}(1)$	$\mathrm{ADD}(\bullet, \mathrm{LIT}(3)) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
1	$\mathrm{ADD}(\bullet, \mathrm{LIT}(3)) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
$\mathrm{LIT}(3)$	$\mathrm{ADD}(1, \bullet) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$

Example

Expression	Context Stack
$\mathrm{ADD}\,(\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(3)), \mathrm{LIT}(5))$	$[\,]$
$\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(3))$	$\mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
$\mathrm{LIT}(1)$	$\mathrm{ADD}(\bullet, \mathrm{LIT}(3)) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
1	$\mathrm{ADD}(\bullet, \mathrm{LIT}(3)) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
$\mathrm{LIT}(3)$	$\mathrm{ADD}(1, \bullet) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
3	$\mathrm{ADD}(1, \bullet) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$

Example

Expression	Context Stack
$\mathrm{ADD}\,(\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(3)), \mathrm{LIT}(5))$	$[\,]$
$\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(3))$	$\mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
$\mathrm{LIT}(1)$	$\mathrm{ADD}(\bullet, \mathrm{LIT}(3)) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
1	$\mathrm{ADD}(\bullet, \mathrm{LIT}(3)) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
$\mathrm{LIT}(3)$	$\mathrm{ADD}(1, \bullet) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
3	$\mathrm{ADD}(1, \bullet) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
4	$\mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$

Example

Expression	Context Stack
$\mathrm{ADD}\,(\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(3)), \mathrm{LIT}(5))$	$[\,]$
$\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(3))$	$\mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
$\mathrm{LIT}(1)$	$\mathrm{ADD}(\bullet, \mathrm{LIT}(3)) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
1	$\mathrm{ADD}(\bullet, \mathrm{LIT}(3)) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
$\mathrm{LIT}(3)$	$\mathrm{ADD}(1, \bullet) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
3	$\mathrm{ADD}(1, \bullet) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
4	$\mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
$\mathrm{LIT}(5)$	$\mathrm{ADD}(4, \bullet) :: [\,]$

Example

Expression	Context Stack
$\mathrm{ADD}\,(\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(3)), \mathrm{LIT}(5))$	$[\,]$
$\mathrm{ADD}(\mathrm{LIT}(1), \mathrm{LIT}(3))$	$\mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
$\mathrm{LIT}(1)$	$\mathrm{ADD}(\bullet, \mathrm{LIT}(3)) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
1	$\mathrm{ADD}(\bullet, \mathrm{LIT}(3)) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
$\mathrm{LIT}(3)$	$\mathrm{ADD}(1, \bullet) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
3	$\mathrm{ADD}(1, \bullet) :: \mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
4	$\mathrm{ADD}(\bullet, \mathrm{LIT}(5)) :: [\,]$
$\mathrm{LIT}(5)$	$\mathrm{ADD}(4, \bullet) :: [\,]$
5	$\mathrm{ADD}(4, \bullet) :: [\,]$

Example

Expression	Context Stack
$\mathrm{ADD}\,(\mathrm{ADD}(\mathrm{LIT}(1),\mathrm{LIT}(3)),\mathrm{LIT}(5))$	$[\,]$
$\mathrm{ADD}(\mathrm{LIT}(1),\mathrm{LIT}(3))$	$\mathrm{ADD}(\bullet,\mathrm{LIT}(5)) :: [\,]$
$\mathrm{LIT}(1)$	$\mathrm{ADD}(\bullet,\mathrm{LIT}(3)) :: \mathrm{ADD}(\bullet,\mathrm{LIT}(5)) :: [\,]$
1	$\mathrm{ADD}(\bullet,\mathrm{LIT}(3)) :: \mathrm{ADD}(\bullet,\mathrm{LIT}(5)) :: [\,]$
$\mathrm{LIT}(3)$	$\mathrm{ADD}(1,\bullet) :: \mathrm{ADD}(\bullet,\mathrm{LIT}(5)) :: [\,]$
3	$\mathrm{ADD}(1,\bullet) :: \mathrm{ADD}(\bullet,\mathrm{LIT}(5)) :: [\,]$
4	$\mathrm{ADD}(\bullet,\mathrm{LIT}(5)) :: [\,]$
$\mathrm{LIT}(5)$	$\mathrm{ADD}(4,\bullet) :: [\,]$
5	$\mathrm{ADD}(4,\bullet) :: [\,]$
9	$[\,]$

Reduction Semantics

Reduction semantics

An alternative representation of small-step operational semantics with explicit representation of the reduction contexts

- The technique is useful to model control operators
- In reduction semantics we have a direct control over what to execute and when

Operational Semantics for μScheme+

Judgment: $\langle e/v, \rho, \sigma, S\rangle \to \langle e'/v', \rho', \sigma', S'\rangle$

- e/v: either an expression e or a value v
- ρ: environment
- σ: store
- S: stack of evaluation context

Assignments

$$\frac{x \in \textit{dom}(\rho)}{\langle \mathrm{SET}(x, e), \rho, \sigma, S\rangle \rightarrow \langle e, \rho, \sigma, \mathrm{SET}(x, \bullet) :: S\rangle}\ \textsc{Assign}$$

$$\frac{\rho(x) = \ell}{\langle v, \rho, \sigma, \mathrm{SET}(x, \bullet) :: S\rangle \rightarrow \langle v, \rho, \sigma\{\ell \mapsto v\}, S\rangle}\ \textsc{Finish-Assign}$$

Conditional

$$\frac{}{\langle \mathrm{IF}(e_1, e_2, e_3), \rho, \sigma, S\rangle \to \langle e_1, \rho, \sigma, \mathrm{IF}(\bullet, e_2, e_3) :: S\rangle}\ \textsc{if}$$

$$\frac{v \neq \mathrm{BOOLV}(\texttt{\#f})}{\langle v, \rho, \sigma, \mathrm{IF}(\bullet, e_2, e_3) :: S\rangle \to \langle e_2, \rho, \sigma, S\rangle}\ \textsc{if}-\textsc{true}$$

$$\frac{v = \mathrm{BOOLV}(\texttt{\#f})}{\langle v, \rho, \sigma, \mathrm{IF}(\bullet, e_2, e_3) :: S\rangle \to \langle e_3, \rho, \sigma, S\rangle}\ \textsc{if}-\textsc{false}$$

`while` loop

$$\frac{}{\langle \text{WHILE}(e_1, e_2), \rho, \sigma, S\rangle \rightarrow \langle e_1, \rho, \sigma, \text{WHILE}(e_1, e_2) :: S\rangle}\ \textsc{while}$$

$$\frac{v \neq \text{BOOLV}(\texttt{\#f})}{\langle v, \rho, \sigma, \text{WHILE}(e_1, e_2) :: S\rangle \rightarrow \langle e_2, \rho, \sigma, \text{EVAL\_BODY}(e_1, e_2) :: S\rangle}\ \textsc{while-true}$$

$$\frac{v = \text{BOOLV}(\texttt{\#f})}{\langle v, \rho, \sigma, \text{WHILE}(e_1, e_2) :: S\rangle \rightarrow \langle \text{BOOLV}(\texttt{\#f}), \rho, \sigma, S\rangle}\ \textsc{while-false}$$

$$\frac{}{\langle v, \rho, \sigma, \text{EVAL\_BODY}(e_1, e_2) :: S\rangle \rightarrow \langle e_1, \rho, \sigma, \text{WHILE}(e_1, e_2) :: S\rangle}\ \textsc{while-body}$$

`try-catch`

(`try-catch` $\mathtt{e}_b$ $\mathtt{e}_h$)

- Evaluate handler $\mathtt{e}_h$ which must produce a function f
- Install f as the most recent handler
- Evaluate body $\mathtt{e}_b$ and proceed as follows:

1. If $\mathtt{e}_b$ evaluates to v without throwing any exception
   - Uninstall the handler f
   - Produce v as the result of the `try-catch` expression
2. If during the evaluation of $\mathtt{e}_b$ a value v is thrown
   - Uninstall the handler f
   - Produce `(f v)` as the result of the `try-catch` expression

`throw` expression

$$\frac{}{\langle \mathrm{THROW}(e), \rho, \sigma, S\rangle \rightarrow \langle e, \rho, \sigma, \mathrm{THROW}(\bullet) :: S\rangle}\ \textsc{throw}$$

$$\frac{}{\langle v, \rho, \sigma, \mathrm{THROW}(\bullet) :: \mathrm{TRYCATCH}(\bullet, v_h) :: S\rangle \rightarrow \langle \mathrm{APPLY}(v_h, v), \rho, \sigma, S\rangle}\ \textsc{throw} - \textsc{transfer}$$

$$\frac{F \text{ does not have the form } \mathrm{TRYCATCH}(\bullet, v)}{\langle v, \rho, \sigma, \mathrm{THROW}(\bullet) :: F :: S\rangle \rightarrow \langle v, \rho, \sigma, \mathrm{THROW}(\bullet) :: S\rangle}\ \textsc{throw} - \textsc{unwind}$$

`try-catch` expression

$$\frac{}{\langle \mathrm{TRYCATCH}(e_b, e_h), \rho, \sigma, S\rangle \to \langle e_h, \rho, \sigma, \mathrm{TRYCATCH}(e_b, \bullet) :: S\rangle}\ \textsc{try-catch}$$

$$\frac{v \text{ is a function}}{\langle v, \rho, \sigma, \mathrm{TRYCATCH}(e_b, \bullet) :: S\rangle \to \langle e_b, \rho, \sigma, \mathrm{TRYCATCH}(\bullet, v) :: \mathrm{LETENV}(\rho) :: S\rangle}\ \textsc{try-catch-handler}$$

$$\frac{}{\langle v, \rho, \sigma, \mathrm{TRYCATCH}(\bullet, v_h) :: S\rangle \to \langle v, \rho, \sigma, S\rangle}\ \textsc{try-catch-finish}$$

Summary

This Lecture

- Reduction Semantics for Control Structures

Next Lecture

- Garbage Collection